CircuitPython

Main Points

1 Background and History

Microcontroller boards are small, self-contained systems with processors, memory, and connections to external peripherals that can be programmed to connect to physical devices. One of the more well-known in microcontroller brands in recent history is *Arduino*. In the words of one of the founders, Massimo Banzi, in his book *Getting Started with Arduino*, "*Arduino* is an open source physical computing platform based on a simple input/output (I/O) board and a development environment that implements the Processing language (www.processing.org). *Arduino* can be used to develop standalone interactive objects or can be connect to software on your computer."

Adafruit Industries (New York, NY) is a manufacturer of a number of US made and sold *Arduino* compatible boards. They modified the design of an *Arduino* board to accept a different processor that can run a version of *Python* forked from *MicroPython*, which was initially developed for the Micro:Bit microcontroller. Adafruit's implementation of *Python* is called *CircuitPython* and is aimed at students and hobbyists, especially those new to electronics and programming. There are currently over 300 *CircuitPython* compatible microcontrollers from Adafruit and other manufacturers.

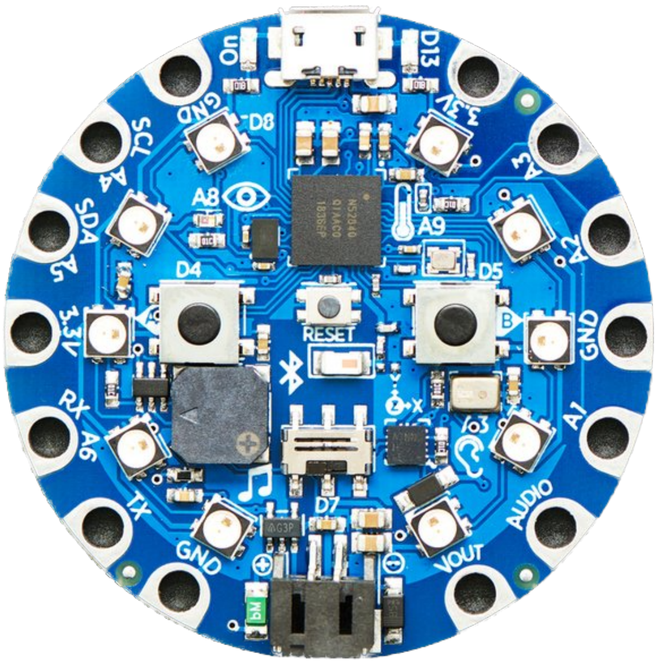
This document is primarily concerned with learning how to use *CircuitPython* to "program the real world" with Adafruit's Circuit Playground Bluefruit. Many of the techniques and operations will also work for other *CircuitPython* compatible boards with little to no modifications. The following links will assist in getting *CircuitPython* onto the board and get up and running and provide a reference to the *CircuitPython* language. Learning to program a microcontroller board may open your mind to possible projects that include machines/devices/systems that need to be controlled and connected to a variety of physical devices/sensors. It may also spark an interest in real world physical computing projects in your everyday life; also known as the Internet of Things (IOT).

- https://learn.adafruit.com/adafruit-circuit-playground-bluefruit/overview
- https://learn.adafruit.com/welcome-to-circuitpython
- https://circuitpython.readthedocs.io/en/latest/?

2 The Circuit Playground Bluefruit (CPB) Hardware

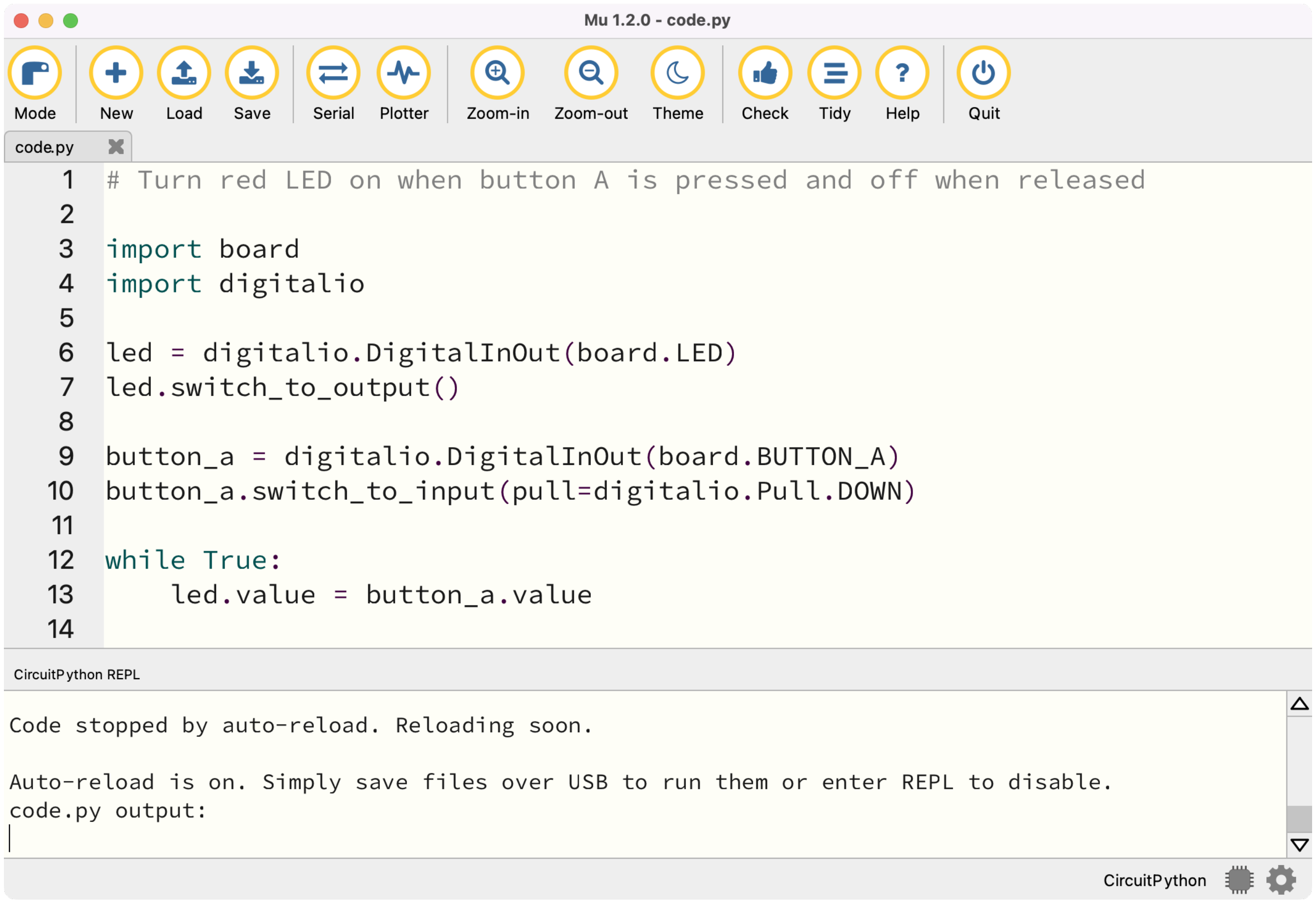
The Adafruit Circuit Playground Bluefruit, aka CPB, is a compact powerhouse of a microcontroller that is well outfitted for hobbyists and students. It includes many built-in devices and sensors. Below is a list of what is on-board the CPB.

- 1 red LED
- 10 multi-color LEDs, aka NeoPixels
- 2 push buttons
- Slide switch
- Temperature sensor
- Light sensor
- 3-Axis accelerometer/motion sensor
- Digital microphone audio sensor
- Speaker (for buzzes and dings and such)
- 14 pads for external connections
  - 6 power-related pads
    - 3 ground (all connected together)
    - 2 3.3V supply connections
    - 1 Vout (5V) supply connection
  - 7 capacitive touch sensors (A1-A6 and TX)
  - 6 analog inputs (A1-A6)
  - 8 digital inputs/outputs (AUDIO, A1-A6, TX)
  - 8 pulse width modulation (PWM) outputs (AUDIO, A1-A6, TX)
- Bluetooth LE (can be paired with a smart phone via an app)



3 Running *CircuitPython* Scripts on the CPB

- Download the *Mu* editor/IDE (or similar CPy aware editor/IDE such as *Thonny*)
- Connect a CPB to the computer with a USB cable
- CPB will show up on the computer as a USB drive named `CIRCUITPY`
- Write/edit and save a script named `code.py` on the CPB (replacing any existing `code.py`)
    - It is strongly suggested that the script be named `code.py`
    - File names `code.txt`, `code.py`, `main.txt`, or `main.py` can be used instead is desired
- Saving the code will cause the board to restart
- The `code.py` script will automatically run when initially connected/powered and after every restart
- Clicking the "Serial" button in *Mu* allows access to...
    - Printed output from running scripts
    - Input command prompts from running scripts
    - Error messages
    - The REPL (command prompt)

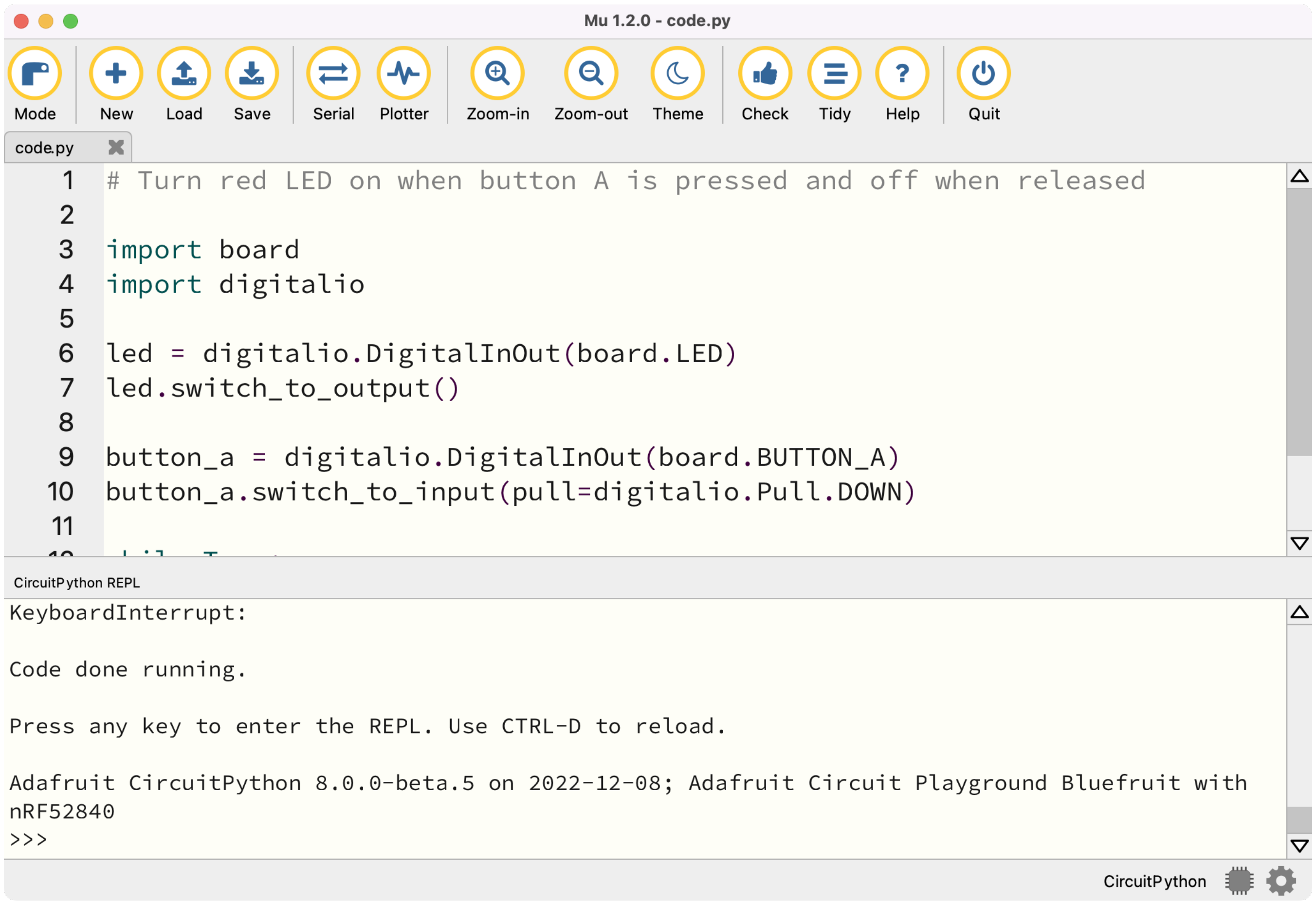


```python
# Turn red LED on when button A is pressed and off when released

import board
import digitalio

led = digitalio.DigitalInOut(board.LED)
led.switch_to_output()

button_a = digitalio.DigitalInOut(board.BUTTON_A)
button_a.switch_to_input(pull=digitalio.Pull.DOWN)

while True:
    led.value = button_a.value

```

```
Code stopped by auto-reload. Reloading soon.

Auto-reload is on. Simply save files over USB to run them or enter REPL to disable.
code.py output:
```

Figure 1: Mu editor with code running with Serial showing

- Activating and using REPL mode
  - Click the “Serial” button to display the REPL
  - Click in the REPL portion of the screen
  - Use the CTRL-C key combination to interrupt the running script
  - Press any key to enter REPL mode and get a command prompt >>>
  - Enter commands as desired (REPL mode does not maintain memory from the script)
  - Use the CTRL-D key combination to leave REPL mode and start running `code.py` again



```
# Turn red LED on when button A is pressed and off when released

import board
import digitalio

led = digitalio.DigitalInOut(board.LED)
led.switch_to_output()

button_a = digitalio.DigitalInOut(board.BUTTON_A)
button_a.switch_to_input(pull=digitalio.Pull.DOWN)

```

```
KeyboardInterrupt:

Code done running.

Press any key to enter the REPL. Use CTRL-D to reload.

Adafruit CircuitPython 8.0.0-beta.5 on 2022-12-08; Adafruit Circuit Playground Bluefruit with
nRF52840
>>>
```

Figure 2: Mu editor showing REPL prompt after CTRL-C and “any” key

4 Some Helpful External *CircuitPython* Libraries to Install

- External libraries may be downloaded and installed on the CPB board
- Download *CircuitPython* libraries from https://circuitpython.org/libraries
- Unzip the download to access the individual libraries/modules
- Drag (or copy) desired libraries/modules to the `lib` folder on the CPB
  - Some of the libraries are folders
  - Others are individual files that have a `.mpy` extension
- Some good libraries for use on the CPB include. . .

- adafruit_ble
- adafruit_bluefruit_connect
- adafruit_bus_device
- adafruit_character_lcd
- adafruit_circuitplayground
- adafruit_debouncer.mpy
- adafruit_displayio_ssd1306.mpy
- adafruit_display_shapes
- adafruit_display_text
- adafruit_hid
- adafruit_lis3dh.mpy
- adafruit_motor
- adafruit_register
- adafruit_ssd1306.mpy
- adafruit_thermistor.mpy
- adafruit_waveform
- neopixel.mpy
- simpleio.mpy

5 *CircuitPython* is Built on *Python*

At its core, *CircuitPython* (CPy) is *Python*. CPy includes support for many of the standard *Python* objects, expressions, statements, commands, and functions, plus adds support for microcontroller specific commands, functions, and modules. Following is a quick review of a number of the *Python* commands that are also valid for *CircuitPython*.

5.1 Object Types, Variables, and Assignment

- Integers, floats, and strings
- F-strings: `f"{5} squared is {5**2}"` will print as `"5 squared is 25"`
- Variables and variable assignment
  - Letters, numbers, and underscores
  - Start with a letter
  - No spaces or special symbols
  - Upper and lowercase are different
  - Example: `num_leds = 10`
- Lists, ranges, and tuples
  - List of numbers: `x = [1, 2, 6, 42, 12]`
  - `range(10)` returns a `range` object with integers 0 through 9
  - `range(1, 10, 2)` returns a `range` object containing `1, 3, 5, 7, 9`
  - Tuple: `position = (3, 4)`
- Dictionaries: `my_dict = {'a': 'eh', 'b': 'bee', 'c': 'see'}`

5.2 Built-in Math

- Basic arithmetic
  - Addition: `4 + 2` returns `6`
  - Subtraction: `10 - 3` returns `7`
  - Multiplication: `12 * 4` returns `48`
  - Division: `16/2` returns `4.0`
  - Exponentiation (raise to a power): `4**2` returns `16`
- Special division operations
  - Integer (floor) division: `7 // 2` returns `3`
  - Remainder division: `7 % 2` returns `1`

- Built-in math functions
    - `pow(4, 2)` returns 16
    - `abs(-100)` returns 100
    - `round(3.14159, 2)` returns 3.14
    - `divmod(5, 2)` returns `(2, 1)`

5.3 The `math` Module

Following is a directory listing of the CPy `math` module from the CPB. Be sure to the `import math` before uwing any of the functions or constants included in it.

Directory of the CPy `math` module

```
Adafruit CircuitPython 8.0.0-beta.5 on 2022-12-08; Adafruit Circuit
↪  Playground Bluefruit with nRF52840
>>> import math
>>> dir(math)
['__class__', '__name__', 'pow', 'acos', 'asin', 'atan', 'atan2',
↪  'ceil', 'copysign', 'cos', 'degrees', 'e', 'exp', 'fabs', 'floor',
↪  'fmod', 'frexp', 'isfinite', 'isinf', 'isnan', 'ldexp', 'log',
↪  'modf', 'pi', 'radians', 'sin', 'sqrt', 'tan', 'trunc']
>>>
```

- Constants
    - Euler's number (e): `math.e`
    - Pi (π): `math.pi`
- General functions
    - Square root: `math.sqrt(5)` $= \sqrt{5}$
    - Natural log: `math.log(7)` $= \ln(7)$
    - Log of any base: `math.log(100, 10)` $= \log_{10} 100$
    - e to a power: `math.exp(3.2)` $= e^{3.2}$
    - Raise to a power: `math.pow(3, 2)` $= 3^2$
- Rounding functions
    - Truncate: `math.trunc(4.13)` returns 4
    - Floor: `math.floor(3.8)` returns 3
    - Ceiling: `math.ceil(4.1)` returns 5
- Trigonometric functions (angles must be in radians)
    - Cosine: `math.cos(math.pi/2)` $= \cos(\pi/2)$
    - Sine: `math.sin(math.pi/2)` $= \sin(\pi/2)$
    - Tangent: `math.tan(math.pi/4)` $= \tan(\pi/4)$
- Inverse trigonometric functions (results are returned in radians)
    - Inverse cosine: `math.acos(4/5)` $= \cos^{-1}(4/5)$
    - Inverse sine: `math.asin(3/5)` $= \sin^{-1}(3/5)$
    - Inverse tangent: `math.atan(3/4)` $= \tan^{-1}(3/4)$
    - Inverse tangent (quadrant aware): `math.atan2(3, -4)` $= \tan^{-1}\left(\frac{3}{-4}\right)$

- Converting between degrees and radians
  - Convert to radians: `math.radians(45)`
  - Convert to degrees: `math.degrees(math.pi/4)`

5.4 Branching, Looping, Functions, and Modules

- Comparison statements
  - Equal to: `4.0 == 2 + 2` returns `True`
  - Not equal to: `5 != 10/2` returns `False`
  - Less than: `2 * 3 < 7` returns `True`
  - Less than or equal to: `3 <= 2**2` returns `True`
  - Greater than: `5 * 5 > 25` returns `False`
  - Greater than or equal to: `42 >= 24` returns `True`
  - Is the same: `5 is 5.0` returns `False`
  - Is not the same: `42.0 is not 42` returns `True`
  - Is a subset of: `'I' in 'team'` returns `False`
  - Is not a subset of: `'I' not in 'team'` returns `True`
- Logical statements (`True` and `False` can be replaced by comparison statements)
  - And: `True and False` returns `False`
  - Or: `True or False` returns `True`
  - Not: `not False` returns `True`
- Conditionals: `if`, `if-else`, and `if-elif-else`
- Looping
  - `for`
  - `while`
  - List comprehensions: `[x**2 for x in range(5)]` creates the list `[0, 1, 4, 9, 16]`
- Function creation
  - User-defined functions using `def`
  - Lambda functions: `(lambda x: x**2 - 2*x +3)(5)` returns 18
- Importing built-in and specialized external modules
  - Use `help('modules')` from the REPL to list standard modules
  - Import a required modules using `import module_name`
  - Download and install external modules in the `lib` folder

Built-in CPy modules on the CPB

```
>>> help('modules')
__main__             board             microcontroller   sharpdisplay
_bleio               builtins          micropython       storage
adafruit_bus_device  busio             msgpack           struct
adafruit_pixelbuf    collections       neopixel_write    supervisor
aesio                countio           onewireio         synthio
alarm                digitalio         os                sys
analogio             displayio         paralleldisplay   terminalio
array                errno             pulseio           time
```

```
atexit              fontio            pwmio             touchio
audiobusio          framebufferio     rainbowio         traceback
audiocore           gc                random            ulab
audiomixer          getpass           re                usb_cdc
audiomp3            gifio             rgbmatrix         usb_hid
audiopwmio          io                rotaryio          usb_midi
binascii            json              rtc               uselect
bitbangio           keypad            sdcardio          vectorio
bitmaptools         math              select            watchdog
```

Plotting using *Matplotlib* is **not** available in *CircuitPython*. *NumPy* is also not available, but there is a *NumPy*-like module called `ulab` that does mimic a subset of *NumPy*. The following code block demonstrates importing `ulab.numpy` and displaying its directory. Not all CPy boards support `ulab`, but the CPB does.

NumPy-like commands for CPy via `ulab`

```
>>> import ulab.numpy as np
>>> dir(np)
['__class__', '__name__', 'all', 'any', 'bool', 'sort', 'sum', 'acos',
↪  'acosh', 'arange', 'arctan2', 'argmax', 'argmin', 'argsort',
↪  'around', 'array', 'asin', 'asinh', 'atan', 'atanh', 'ceil', 'clip',
↪  'compress', 'concatenate', 'convolve', 'cos', 'cosh', 'cross',
↪  'degrees', 'diag', 'diff', 'dot', 'e', 'empty', 'equal', 'exp',
↪  'expm1', 'eye', 'fft', 'flip', 'float', 'floor', 'frombuffer',
↪  'full', 'get_printoptions', 'inf', 'int16', 'int8', 'interp',
↪  'isfinite', 'isinf', 'linalg', 'linspace', 'log', 'log10', 'log2',
↪  'logspace', 'max', 'maximum', 'mean', 'median', 'min', 'minimum',
↪  'nan', 'ndarray', 'ndinfo', 'not_equal', 'ones', 'pi', 'polyfit',
↪  'polyval', 'radians', 'roll', 'set_printoptions', 'sin', 'sinh',
↪  'sqrt', 'std', 'tan', 'tanh', 'trace', 'trapz', 'uint16', 'uint8',
↪  'vectorize', 'where', 'zeros']

>>> dir(np.linalg)
['__class__', '__name__', 'cholesky', 'det', 'eig', 'inv', 'norm', 'qr']

>>> np.arange(0, 5.1, 0.5)
array([0.0, 0.5, 1.0, ..., 4.0, 4.5, 5.0], dtype=float32)
>>> np.linspace(0, np.pi, 10)
array([0.0, 0.349066, 0.698132, 1.0472, 1.39626, 1.74533, 2.09439,
↪  2.44346, 2.79253, 3.14159], dtype=float32)
```

6 Commonly Used *CircuitPython* Modules

Most of the fundamental programming concepts that can be used with *Python* can also be used when programming with *CircuitPython*. The biggest difference is the addition of special modules for interacting with external devices. Additional modules may be imported to do things like...

- Access physical objects on the micro-controller board
- Turn on/off lights, motors, etc.
- Read the input from buttons, switches, sensors, etc.

The following block shows a number of possible module imports that are available. Each will be discussed in more detail.

Commonly used CPy-specific modules to import

```
# Nearly all CPy boards
import board        # access to board objects, such as I/O pins
import time         # time functions
import digitalio    # on/off inputs/outputs
import analogio     # variable inputs/outputs
import pulseio      # pulsed inputs/outputs
import simpleio     # miscellaneous input/output support
import neopixel     # access and control of multicolored LEDs
import random       # random numbers
```

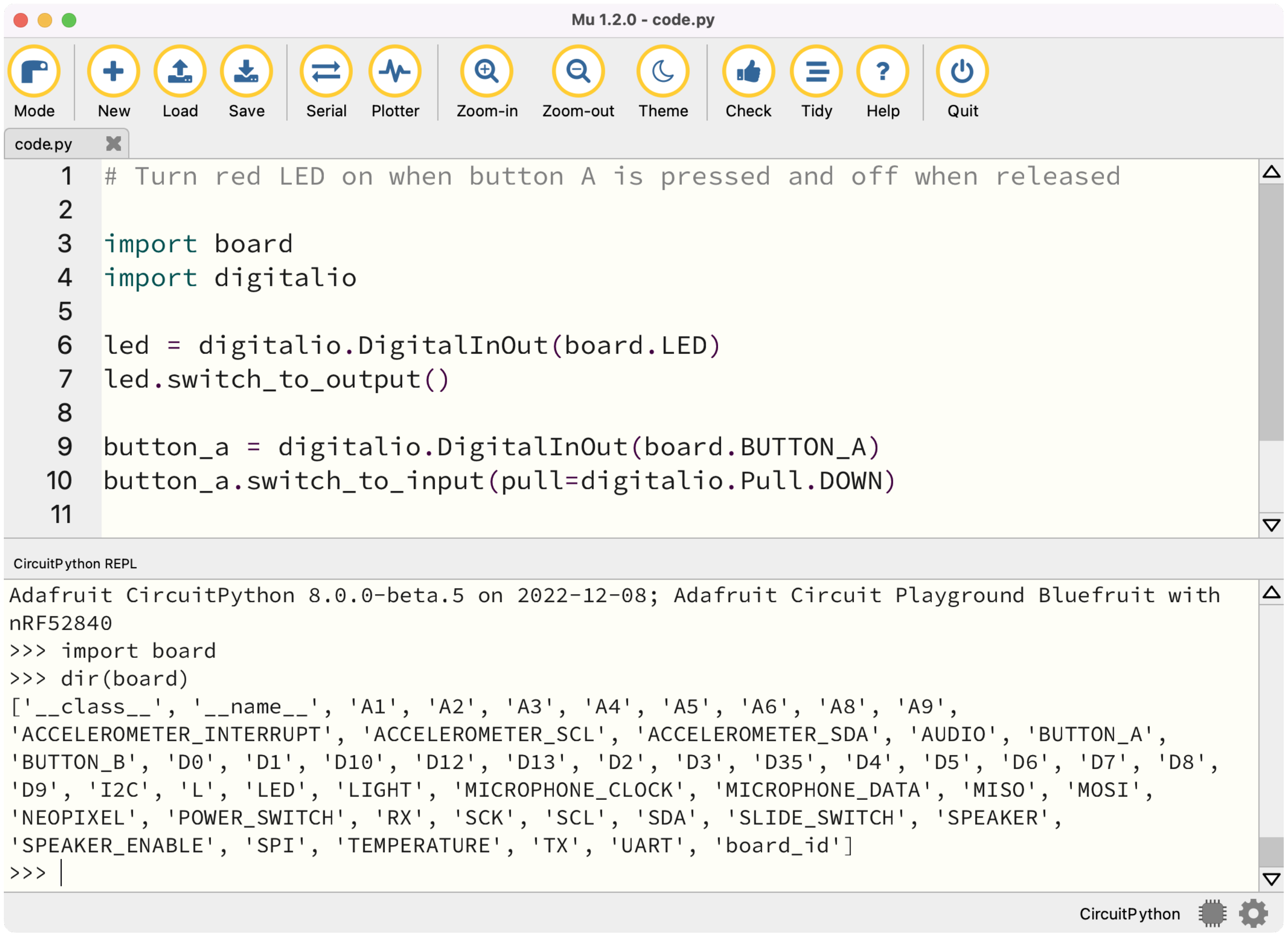
The Circuit Playground Bluefruit has an additional and very powerful helper module that can be used as an alternative to some of the above modules. The functions and commands in this helper module will be discussed in a later section.

- Import via the command `from adafruit_circuitplayground import cp`
- Allows for easier interaction with the many built-in devices and sensors
- No special setup commands are required for on-board devices when using `cp`

6.1 The `board` Module

The `board` module provides access to the available board pins/pads and interface names for objects on the device.

- Examples...
  - `board.D0` refers to digital input/output (I/O) pin `0`
  - `board.A0` refers to analog input pin `0`
- Descriptive variable names can be assigned to board-level names...
  - `button_pin = board.D0`
  - `sensor_pin = board.A0`
- Use `dir(board)` from the REPL prompt to see the supported `board` object names



```python
# Turn red LED on when button A is pressed and off when released

import board
import digitalio

led = digitalio.DigitalInOut(board.LED)
led.switch_to_output()

button_a = digitalio.DigitalInOut(board.BUTTON_A)
button_a.switch_to_input(pull=digitalio.Pull.DOWN)

```

```
Adafruit CircuitPython 8.0.0-beta.5 on 2022-12-08; Adafruit Circuit Playground Bluefruit with
nRF52840
>>> import board
>>> dir(board)
['__class__', '__name__', 'A1', 'A2', 'A3', 'A4', 'A5', 'A6', 'A8', 'A9',
'ACCELEROMETER_INTERRUPT', 'ACCELEROMETER_SCL', 'ACCELEROMETER_SDA', 'AUDIO', 'BUTTON_A',
'BUTTON_B', 'D0', 'D1', 'D10', 'D12', 'D13', 'D2', 'D3', 'D35', 'D4', 'D5', 'D6', 'D7', 'D8',
'D9', 'I2C', 'L', 'LED', 'LIGHT', 'MICROPHONE_CLOCK', 'MICROPHONE_DATA', 'MISO', 'MOSI',
'NEOPIXEL', 'POWER_SWITCH', 'RX', 'SCK', 'SCL', 'SDA', 'SLIDE_SWITCH', 'SPEAKER',
'SPEAKER_ENABLE', 'SPI', 'TEMPERATURE', 'TX', 'UART', 'board_id']
>>> 
```

Figure 3: Mu REPL showing directory of `board`

The built-in devices are also generally associated with more than one `board` item.

- BUTTON_A (D4): momentary push button
- BUTTON_B (D5): momentary push button
- SLIDE_SWITCH (D7): "on/off" slider switch
- NEOPIXEL (D8): 10 multi-color LEDs
- LIGHT (A8): light sensor
- TEMPERATURE (A9): temperature sensor
- L or LED (D13): small red LED

Most of the pads around the perimeter of the CPB are associated with more than one name in the `board` list.

- AUDIO (D12 or SPEAKER): use as a digital I/O or PWM out (more on PWM later)
- A1 (D6): use as an analog input, digital I/O, PWM out, or capacitive touch
- A2 (D9): use as an analog input, digital I/O, PWM out, or capacitive touch
- A3 (D10): use as an analog input, digital I/O, PWM out, or capacitive touch

- A4 (D3): use as an analog input, digital I/O, PWM out, or capacitive touch
- A5 (D2): use as an analog input, digital I/O, PWM out, or capacitive touch
- A6 (RX or D0): use as an analog input, digital I/O, PWM out, or capacitive touch
- TX (D1): use as an analog input, digital I/O, PWM out, or capacitive touch

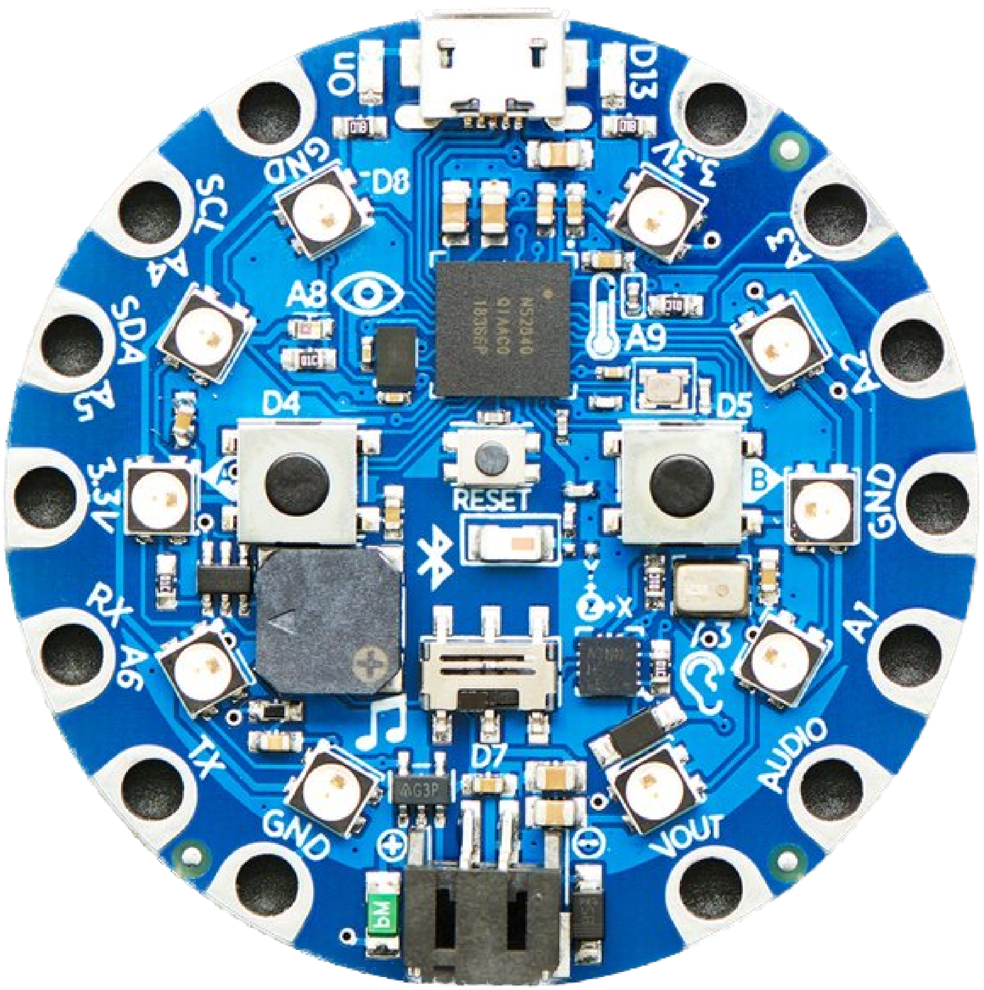


Figure 4: The Circuit Playground Bluefruit (CPB) board

6.2 The `digitalio` Module

The `digitalio` module is used for interacting with on/off style inputs and outputs, such as simple push buttons and LED lights. The `board` and `digitalio` modules must be imported before initialization can occur.

- `digitalio.DigitalInOut()` initializes a pin to be used as a digital input or output
- The following code blocks demonstrate setting up `board.D0` as a digital input named `button1`
- All digital I/O pins default to the `INPUT` state when initialized

Setting up a digital input – method 1

```
import board
import digitalio
button1 = digitalio.DigitalInOut(board.D0)
```

Setting up a digital input – method 2

```
import board
import digitalio
button1 = digitalio.DigitalInOut(board.D0)
button1.switch_to_input()   # optional for inputs
```

Setting up a digital input – method 3

```
import board
import digitalio
button1 = digitalio.DigitalInOut(board.D0)
button1.direction = digitalio.Direction.INPUT  # optional for inputs
```

- The next code blocks demonstrate initializing `board.D13` as a digital output named `LED1`
- On the CPB `board.D13` is assigned to an on-board red LED
- Initializing a pin as an output always requires setting a direction
- The direction can be set using either of the methods shown

Setting up a digital output – method 1

```
LED1 = digitalio.DigitalInOut(board.D13)
LED1.switch_to_output()
```

Setting up a digital output – method 2

```
LED1 = digitalio.DigitalInOut(board.D13)
LED1.direction = digitalio.Direction.OUTPUT
```

External circuits with buttons/switches or LEDs usually need to incorporate resistors. Resistors used with LEDs keep the LEDs from burning out due to too much current. Resistors used with buttons/switches "anchor" the voltage to ground or high voltage when the button is not activated. Such resistors are called pull-up or pull-down resistors and are included on the CPB for the built-in buttons and slide switch, but need to be activated to be used. The built-in push buttons have pull-down resistors and the built-in slide switch has a pull-up resistor.

Activating built-in pull-up or pull-down resistors

```
# button_a will be False when not pressed and True when pressed
button_a = digitalio.DigitalInOut(board.BUTTON_A)
button_a.switch_to_input(pull=digitalio.Pull.DOWN)

# slider will be False when slid right and True when slid left
slider = digitalio.DigitalInOut(board.SLIDE_SWITCH)
slider.switch_to_input(pull=digitalio.Pull.UP)
```

- Read the state of digital inputs by using the `.value` method
  - `button_a.value` results in `True` when pressed in the above case
  - `button_a.value` results in `False` when not pressed in the above case
  - `slider.value` results in `True` when slid left in the above case
  - Reading the state is typically done as part of an `if` statement (see code block)
  - Depending upon how an input is wired, `True` could mean either pressed or not pressed
- Setting the state of an output, i.e. `LED1`, to either on or off by assigning its value to either...
  - `True` for on, i.e. `LED1.value = True`
  - `False` for off, i.e. `LED1.value = False`

The following code will turn on `LED1` when `button_a` is `True` (assuming both devices had previously been initialized). Any time `button_a` is `False` so will `LED1`.

Setting the state of an LED to match the state of a button

```
while True:
    if button_a.value:   # same as using button_a.value == True
        LED1.value = True
    else:
        LED1.value = False
```

A more *Pythonic* approach is shown below.

- The state of `LED1` is set to be the same as the state of `button_a`
- When `button_a.value` is `True`, then `LED.value` is `True`

A Pythonic way to turn on an LED

```
while True:
    LED1.value = button_a.value
```

More on Resistors with Buttons

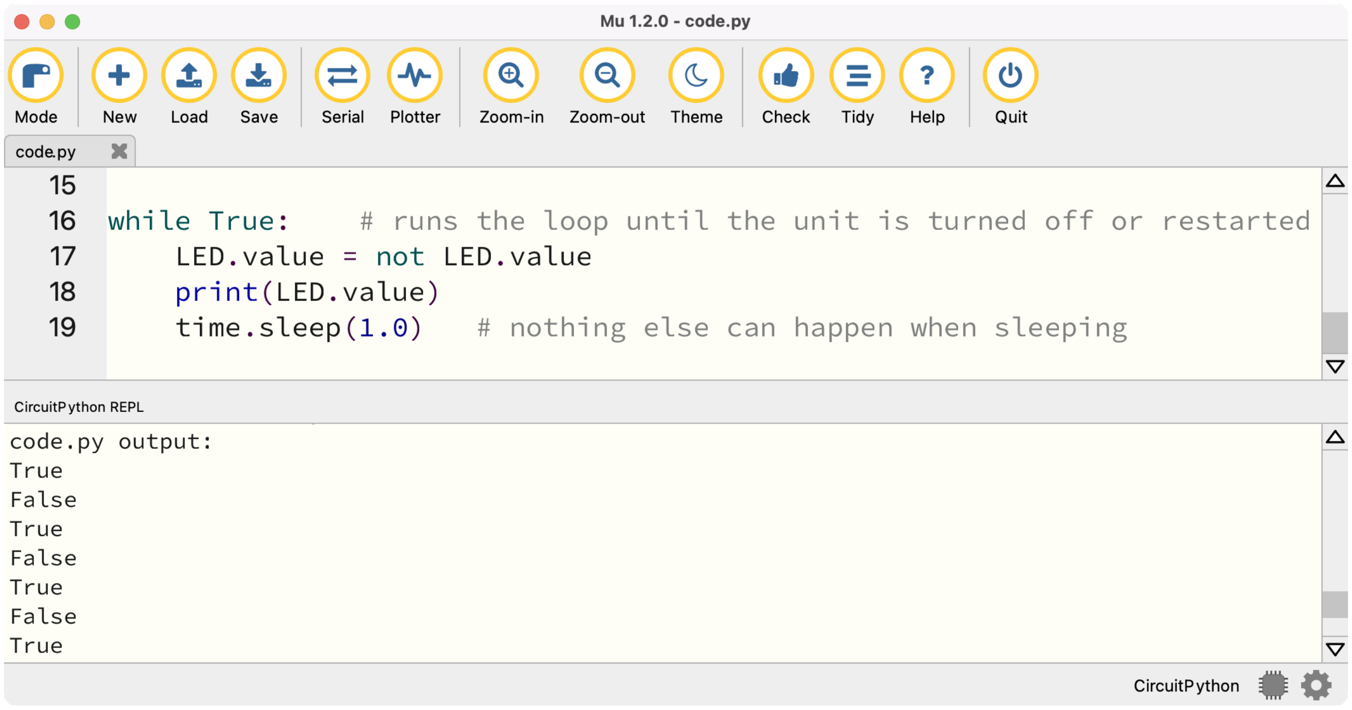
- Whether a switch is `True` or `False` when it is pressed or not depends upon how it is wired
- Using a pull-up resistor with an external button named `button1`
  - One leg of the switch is connected to the input pin with an external resistor also connected from that leg to $3.3V$
  - Other leg of the switch is connected to ground
  - When not-pressed, the voltage is pulled-up to 3.3V and `button1.value` will be `True`
  - When pressed, `button1.value` will be `False`
- The CPB includes internal pull-up resistors at each of digital pins
  - When using an internal resistor, you must not use an external resistor as well
  - Using the internal pull-up resistor will make a button act the same as above
  - If using the internal resistor, one leg of the switch is wired to the digital pin and the other leg to ground
  - After setting the pin as an input in the code, add either of the following lines (not both) to make use of the internal resistor
    - `button1.pull = digitalio.Pull.UP`
    - `button1.switch_to_input(pull=digitalio.Pull.UP)`
- Using a pull-down resistor with a external button named `button1`
  - One leg of the switch is connected to the input pin with an external resistor also connected from that leg to ground
  - Other leg of the switch is connected to 3.3V
  - When not-pressed the voltage is pulled-down to ground and `button1.value` will be `False`
  - When pressed, `button1.value` will be `True`
- Using the internal resistors with switches (push buttons) requires...
  - A little bit more coding
  - A little less wiring
- Regardless of which method is chosen, care should be taken to not mix them together for the same input connection

Digital IO reference: https://learn.adafruit.com/circuitpython-digital-inputs-and-outputs

The next code block initializes an LED as a digital input and blinks it continuously at the rate of one second and one second off. This code should work on any *CircuitPython* board. By using `True` for the `while` comparison expression the loop will run until the unit is turned off or restarted. This is a common approach when programming microcontrollers.

Simple code.py file to blink the red LED

```
"""
Use the Serial button in Mu to see the printed value of the light status,
where True is on and False is off. Change the rate of blinking (in
seconds) by changing the value in `time.sleep()`
"""

import board
import digitalio
import time

LED = digitalio.DigitalInOut(board.D13)
LED.switch_to_output()

LED.value = False  # turn the LED "off" to start

while True:    # runs the loop until the unit is turned off or restarted
    LED.value = not LED.value
    print(LED.value)
    time.sleep(1.0)   # nothing else can happen when sleeping
```



```

while True:    # runs the loop until the unit is turned off or restarted
    LED.value = not LED.value
    print(LED.value)
    time.sleep(1.0)   # nothing else can happen when sleeping
```

```
code.py output:
True
False
True
False
True
False
True
```

Figure 5: Mu editor showing REPL with printed code output

This next code sample sets the value of the LED to match the value of the slide switch. When the switch is `True` the LED will be `True` (on).

A script to turn on/off the LED with the slide switch

```
import board
import digitalio

slider = digitalio.DigitalInOut(board.SLIDE_SWITCH)
slider.switch_to_input(pull=digitalio.Pull.UP)

red_LED = digitalio.DigitalInOut(board.L)
red_LED.direction = digitalio.Direction.OUTPUT

while True:
    red_LED.value = slider.value
```

6.3 The `time` Module

The `time` module includes two time-related functions that are regularly used; `time.sleep()` and `time.monotonic()`.

- `time.sleep(x)`
  - Delay execution of the script by `x` seconds
  - Execution restarts at the next line of code
  - `time.sleep(1.5)` creates a 1.5 second delay
  - Outputs will retain their current states while sleeping
- `time.monotonic()`
  - Used for relative timing (the CPB does not keep real clock times)
  - Does not accept any arguments
  - Returns a float associated with the time in seconds since the last device reset
  - Allows timing between events by comparing the current value of `time.monotonic()` to a previously saved one
  - Timed events can occur without stopping the execution of other lines of code when using `time.monotonic()`

The following example shows how `time.monotonic()` might be used for blinking an LED without stopping all other code execution (aka blocking). This code changes the state of the LED every 3 seconds without using `time.sleep()`. The LED is turned on (or off) and the current `time.monotonic()` value is saved to the variable `time_old` before the `while True` loop. The current value of `time.monotonic()` is compared to the saved value each iteration of the loop to determine if 3 seconds have elapsed. Once 3 seconds have passed, the state of the LED is changed to the opposite state and the variable holding the previous `time.monotonic()` value is updated to the new time.

Blinking an LED using `time.monotonic()`

```
import time
import board
import digitalio

LED1 = digitalio.DigitalInOut(board.D13)  # built-in red LED
LED1.switch_to_output()

time_old = time.monotonic()   # set the start time to the "clock" time
LED1.value = True

while True:
    # check to see if at least 3 seconds have passed
    if time.monotonic() >= time_old + 3:
        LED1.value = not LED1.value
        # reset the start time to the new "clock" time
        time_old = time.monotonic()
```

START HERE

6.4 The `analogio` Module

The `analogio` module provides commands/functions for working with analog inputs and outputs. Analog electrical devices either provide a varied voltage related to a physical phenomenon (analog inputs) or require voltages from within a specified range for operation (analog outputs). The analog input commands/functions are used to read variable signals from analog devices such as potentiometers, temperature sensors, light sensors, and flex sensors. Analog outputs (if supported) may be used to provide a range of voltage values to a motor to vary its speed or a very specific voltage value to an electrical circuit to ensure correct operation.

- The voltage from analog sensors often varies linearly from 0V up to either 3.3V or 5V
- CPB (and nearly all CPy boards) use 3.3 V as the maximum analog voltage level
- Voltage divider (or similar) circuits can be used to reduce sensor outputs todesired levels
- An analog to digital converter (ADC) converts analog input voltages into integer values for use within a script
  - All Adafruit boards divide the low to high voltage signals into 16-bits of data (0 to 65535)
  - 0 V equates to an internal value of `0`
  - 3.3 V equates to an internal value of $2^{16} - 1$ or 65535
- The ADC on the CPB is a 12-bit device, but converts values to 16-bits for programming purposes
- All analog pins in the `board` module start with the letter `A`
- The CPB includes six analog pins (pads actually); `board.A1` through `board.A6` plus `board.TX`

The following example code demonstrates how to initialize `board.A1` as an analog input and read its value. Initialization only needs to be done once. However, the value from the analog input will need to be read each it is needed within a script. Reading the value is usually done within the main `while` loop.

Initializing an analog input then reading its value

```
sensor = analogio.AnalogIn(board.A1) # done once early in the script

sensor.value  # use any time the sensor reading is required
```

This next example shows how to convert an analog value back to a voltage. Since the standard reference voltage for the CPB is 3.3 V, it can be used in the calculation shown first. The `.reference_voltage` property can be requested for an analog input to check the reference voltage level used. The second calculation uses the reference voltage reading. In most cases, both should return the same value.

Converting from an analog input reading to a voltage

```
sensor.value / 65535 * 3.3

# Or use the reference voltage for the pin
sensor.value / 65535 * sensor.reference_voltage
```

- The CPB does not have a true analog output pin/pad
- The CPB has `AUDIO` to provide for audio output to the internal speaker or an external device
- Some CPy boards have a true analog output pin
  - Their digital to analog converters (DACs) usually support 10-bit resolution
  - For consistency sake, 16-bits are used in the programming environment just like for inputs
  - The analog output pin on the M0 Express board is `board.A0`

Setting the value of a true analog output (not available on the CPB)

```
var_output = analogio.AnalogOut(board.A0)

var_output.value = 50000
```

Below is a short script that demonstrates the reading of an analog input. The example assumes a light sensor, such as a photoresistor, is connected to the A2 pad on the CPB as an analog input. As the light intensity reaching the sensor increases/decreases the voltage at the input will change accordingly. A calculation is included that converts (or scales) the analog input value from a 0 to 65535 range to a 0 to 10 range.

Short script to read and print values from an external light sensor

```
import time
import board
import analogio

light_sensor = analogio.AnalogIn(board.A2)

while True:
    light_value = light_sensor.value
    # scale the light_level to a range of 0-10
    light_level*10/65535
    print(f"The light level is {light_level}")
    time.sleep(2)
```

Analog reference: https://learn.adafruit.com/circuitpython-basics-analog-inputs-and-outputs

6.5 The `pwmio` Module

Although the CPB boards do not include analog outputs, they do provide for pulse width modulation (PWM) outputs. PWM can be used in some cases to act like an analog output, not by varying the voltage, but by pulsing the output between high and low at a particular frequency.

- All of the digital I/O pins on the CPB can be configured as PWM outputs
- The `PWMOut` function from the `pwmio` module is used for PWM outputs
- PWM outputs can be used with various devices
    - Change apparent intensity of LEDs
    - Vary speeds of hobby motors
    - Change position of hobby servos
    - Drive small speakers/buzzers to make various tones
    - Signal to external motor driver boards for DC, stepper, and servo motors

The examples in the following code block initialize pin/pad `D6` as a PWM output named `motor` and set the value using the `.duty_cycle` property. Duty cycle values must be integers from `0` (off) to `65535` (fully on) for any PWM output. The last two lines set a variable to be a value between 0 and 100 (i.e. a percentage of full speed) and use that variable to set the duty cycle.

Initializing a PWM output and setting its duty cycle

```
motor = pwmio.PWMOut(board.D6)
motor.duty_cycle = 42500

percentage = 66
motor.duty_cycle = int(percentage / 100 * 65535)
```

Example code below increases and decreases the intensity of the built-in red LED (board object `D13` or `L`) by small amounts over time such as to create a slow "blinking" effect. A pair of `for` loops are used to make the incremental changes; one loop for increasing the intensity and the other for decreasing it.

Change intensity the red LED using PWM

```
import time
import board
import pwmio

red_LED = pwmio.PWMOut(board.L)
red_LED.duty_cycle = 0      # start with the light off

while True:
    for intensity in range(0, 65536, 64): # low to high
        red_LED.duty_cycle = intensity
        time.sleep(0.005)
    for intensity in range(65535, -1, -64): # high to low
        red_LED.duty_cycle = intensity
        time.sleep(0.005)
```

6.6 The `simpleio` Module

The `simpleio` module provides for a number of specialized helper commands/functions including the following...

- `tone()`
  - The `tone(pin, frequency, duration)` function will generate a tone to play via a PWM output
  - The frequency argument is in hertz and the duration is in seconds
  - The duration argument is optional; if not used it defaults to 1 second
- `bitWrite()`, `shift_in()`, and `shift_out()` to perform bit-level operations
- `map_range()`
  - The `map_range(x, in_min, in_max, out_min, out_max)` function scales `x` from one range to another
  - The returned value from `map_range()` is always a float
  - For example, `map_range(5000, 0, 65535, 0, 10)` scales the starting value `5000`
    - From a of range of `0` to `65535`
    - To a new range of `0` to `10`
    - The return value in this example will be approximately `0.76295`
  - This function is helpful when input devices use a different low-high range than output devices
- `DigitalOut()` and `DigitalIn()`
  - Duplicate functionality from the `digitalio` module
  - Simplifies the initialization of digital inputs and outputs

The `simpleio.map_range()` function is demonstrated in the following code block. It is used to scale the value of a light sensor analog input from 0-65535 to 0-10. The retured value form the function is rounded to an integer value.

Using `map_range()` to convert an analog reading to a 0-10 integer value

```
import board
import time
from simpleio import map_range
import analogio

light_sensor = analogio.AnalogIn(board.LIGHT)

# Read, scale, and print a 0 to 10 light level every second
while True:
    light_raw = light_sensor.value
    light_scaled = round(map_range(light_raw, 0, 65535, 0, 10))
    print(f"Light level: Raw = {light_raw}, scaled = {light_scaled}")
    time.sleep(1.0)
```

The next code block uses the `DigitalIn()` and `DigitalOut()` functions from the `simpleio` module to set up a button and LED. The `simpleio` module imports `digitalio` locally to perform the required tasks. This example also sets a state of the LED and toggles it every time the button is pressed instead of having to have the button pressed for the LED to remain lit.

Example using `simpleio.DigitalOut()` and `simpleio.DigitalOut()`

```
import board
import time
import simpleio

LED2 = simpleio.DigitalOut(board.D3)
button2 = simpleio.DigitalIn(board.D4)

# code below this line is same whether simpleio or digitalio is used

on_off = False                  # sets initial state of on_off
while True:
    if button2.value:           # assuming pressed is True
        on_off = not on_off     # toggles state of on_off
        LED2.value = on_off     # set state of LED to new on_off value
        time.sleep(0.2)         # give some time to get finger off button
```

6.7 The `random` Module

The `random` module is used to generate pseudo-random numbers for script testing, game play, and other purposes.

- `random.random(x)`: a random float between 0 and 1.0, not including 1.0
- `random.randrange(stop)`: a random integer between 0 and `stop`, not including `stop`
- `random.randrange(start, stop)`: a random integer between `start` and `stop`, not including `stop`
- `random.randrange(start, stop, step)`: a random integer between `start` and `stop`, not including `stop`, with the interval `step`
- `random.randint(start, stop)`: a random integer between `start` and `stop`, including `start` and `stop`
- `random.choice(x)` for random item from sequence `x` (usually a list)
- `random.uniform(start, stop)`: a float from a uniform distribution between `start` and `stop`, including `start` and `stop`

6.8 The `displayio` Module

The `displayio` module and some of its related the modules can be used to send text, images, and shapes to external displays. The MECH 322 course kits include an Adafruit 0.91 inch, 128x32 pixel monochrome OLED display that communicates with the CPB via the I^2C protocol via a cable with a Stemma QT connector on one end for connecting to the display and banana clips on the other end for connecting to I2C on the CPB. The `terminalio`, `adafruit_display_text`, and `adafruit_displayio_ssd1306` modules are used in conjunction with `displayio` and the OLED display.

The Stemma connector wires are color-coded as...

- Red = 3.3V
- Black = ground
- Blue = SDA
- Yellow = SCL

A fair amount of code is required to setup, initialize, and update the display with *CircuitPython*. Thhe code that follows is a bit of annotated code for initializing the display, setting up three text lines, and placing initial strings on each line. The third line of the display is then updated each time through the `while True` loop. The general steps that are needed when using the display board are:

1. Import the required modules
2. Release all existing connected displays
3. Use `displayio.I2CDisplay()` to initialize and name the hardware connection
    - `display_bus` is the name used in the sample code
    - Connection type is `board.I2C()` with `device_address=0x3c`
4. Create and name a display object using `adafruit_displayio_ssd1306.SSD1306()`
    - Display object is named `display` in the sample code
    - Uses the hardware connection named `display_bus`
    - Display size set to `width=128` and `height=32`

5. Set the font to be used for text; `terminalio.FONT` in this case
6. Create `adafruit_display_text.label.Label` objects for each line of text
   - The name `text_line` is used in the sample code
   - A list with three objects, one for each line of text, is used here
   - Using `terminalio.FONT` yields 21 characters per line with the sample code settings
7. Create and name a `displayio.Group` to place the text objects into
   - The name `text_display` is used as the group name in the sample code
   - The sample group is located at the upper left corner of the display; (0,0)
8. Use `display.show()` to push the `displayio.Group` object to the physical display
9. Update the text in each `adafruit_display_text` object using the `.text` property
   - For example, `text_display[0].text = "Hello"` will update the top line
   - `text.display[1].text = "Python".center(21)` centers `Python` on the middle line
10. Use string functions and objects to create the desired display text

Steps 1-8 (or similar) are needed for any script that uses the display board from the kit. The sample code for display initialization can be copied and pasted into any `code.py` script. Essentially, all of the lines before `while True:` except the `time.sleep()` command are used to initialize the display. Only the starting text needs to be changed. The number of items in `text_line` and the x and y start values can be changed if only one or two lines of text are desired. The text size can also be scaled by integer amounts.

Setting up an external display for text using `displayio`

```
"""
Stemma QT to alligator clips
red:    3.3V
black:  GND
blue:   SDA
yellow: SCL
"""

# import modules needed for using the oled display
import board
import displayio
import terminalio
import adafruit_displayio_ssd1306
from adafruit_display_text import label

# other imports
import time

# release all existing displays (otherwise errors on restart)
displayio.release_displays()
```

```
#====== Configure display hardware ======#
# use I2C with a specific address (printed on back of board)
display_bus = displayio.I2CDisplay(board.I2C(), device_address=0x3c)

# each display board series uses a specific library/module
display = adafruit_displayio_ssd1306.SSD1306(display_bus,
                                             width=128
                                             height=32)

#====== Configure the text display ======#
# assign the terminalio font for this case
font = terminalio.FONT

# create 3 text 'adafruit_display_text.label' objects in a list
# font is 8 pixels high and display is 32 pixels high
# each line of text is initialized with 21 blank characters
# maximum number of characters with the terminalio font is 21
text_line = []
for y in [3, 13, 25]:
    text_line.append(label.Label(font=font,
                                 text=' ' * 21,
                                 color=(255, 255, 255),
                                 x=2, y=y))

# create a 'displayio.Group' object named 'text_display'
# to hold the 3 lines of text that starts at the upper, left
# corner of the screen (0, 0), then show 'text_display'
text_display = displayio.Group(x=0, y=0)
display.show(text_display)

# initialize 3 lines of text and append them to the group object
# 21 characters maximum for each line
line = ["*" * 21, "-" * 21, "=" * 21]
for i in range(3):
    text_line[i].text = line[i]
    text_display.append(text_line[i])

# the display has now been initialized with text on all 3 lines
# use text_line[<i>].text = <string> to update the text on any line
# for example, text_line[0].text = "Hello, World!"
time.sleep(2)  # hold the initial text on the screen for 2 seconds

while True:
    # change the text on a line by the updating its '.text' property
    # the following line clears the last line (index position 2)
    text_line[2].text = ''

    for i in range(20):
        # this option would display 12345678901234567890
        # on the bottom line adding one digit per iteration
        # text_line[2].text += str((i+1) % 10)

        # this option would display a '*' that moves across the
        # screen on the bottom line one character space per iteration
        # text_line[2].text = " "*i + "*"

        # this option will display *'s across the screen on
        # the bottom line that equal the iteration value + 1
        text_line[2].text = "*" * (i + 1)
        time.sleep(0.5)
```

7 The `adafruit_circuitplayground` Helper Module for CPB

Adafruit created a helper module specifically for the Circuit Playground series of boards, of which, the CPB is the newest and fastest of the series. This module, `adafruit_circuitplayground`, makes it much easier to access and use the built-in hardware on the CPB. This module requires that other specific modules be installed and available in the `lib` folder on the `CIRCUITPY` drive. Because of this, the `adafruit_circuitplayground` module requires more drive space and memory. However, the CPB board has more than enough resources for this to not be an issue in most cases.

7.1 Required Modules in the `lib` Folder

The `adafruit_circuitplayground` module includes code to import the modules required to perform its tasks (see the following code block). `code.py` cannot access these modules unless explicitly imported Imports made by the `adafruit_circuitplayground` module are only available locally to the module.

Imports performed by `adafruit_circuitplayground`

```
import math
import array
import time
try:
    import audiocore
except ImportError:
    import audioio as audiocore
import adafruit_lis3dh
import adafruit_thermistor
import analogio
import board
import busio
import digitalio
import neopixel
import touchio
import gamepad
import audiopwmio
import audiobusio
```

Of the above modules, only the following are not part of the standard *CircuitPython* library and need to be installed in the `lib` folder of `CIRCUITPY`

- `adafruit_bus_device`
- `adafruit_lis3dh.mpy`
- `adafruit_thermistor.mpy`
- `neopixel.mpy`

7.2 Importing the `adafruit_circuitplayground` Module

- Use either of the following commands to import this module for use on the CPB
- The first is the easier and newer method, but the second may still show up in some tutorials

Options for importing `adafruit_circuitplayground` on the CPB

```
from adafruit_circuitplayground import cp
# from adafruit_circuitplayground.bluefruit import cpb
```

7.3 Available `cp` Commands

Following are most of the commands available after importing `adafruit_circuitplayground` as `cp`. All of the commands need to start with `cp.` to access the commands from the imported module.

- **Buttons, slide switch, and red LED**
  - `cp.button_a` returns the value of `board.BUTTON_A`
    - Pressed is `True`
    - Not pressed is `False`
  - `cp.button_b` returns the value of `board.BUTTON_B`
    - Pressed is `True`
    - Not pressed is 'False'
  - `cp.were_pressed` returns a set containing which buttons have been pressed since the last request

    - `cp.switch` returns the value of `board.SLIDE_SWITCH`
        - Left is `True`
        - Right is `False`
    - `cp.red_led` sets the value of the red LED to `True` or `False`, i.e. `cp.red_led = True`
- **NeoPixels**
    - `cp.pixels` works with the 10 multi-colored LEDs (NeoPixels) referenced by `board.NEOPIXEL`
    - `cp.pixels` acts much like a list that is indexed starting with 0
    - `cp.pixels.n` returns the number of NeoPixels
    - `cp.pixels.auto_write` can be assigned either `True` or `False`
        - If `cp.pixels.autor_write = False`, then `cp.pixels.show()` is required to enable changes made to the NeoPixels
        - If `True` (which is the default), changes to colors or brightness occur as soon as values are assigned
    - `cp.pixels.brightness`
        - `cp.pixels.brightness = 0` is off
        - `cp.pixels.brightness = 1` is the brightest
        - `cp.pixels.brightness = 0.3` is 30% brightness
    - `cp.pixels[n] = color` sets the color of the NeoPixel at index position `n`
        - Can use the tuple `(red, green, blue)` to set `color`
            - Values for each color range from 0 to 255
            - Red is `(255, 0, 0)`, green is `(0, 255, 0)`, and blue is `(0, 0, 255)`
        - Can also use a hex code to set `color`
            - Six digit hex value, for example `#ba0c2f` is Ferris crimson
            - First 2 digits = red part, second 2 digits = green part, and third 2 digits = blue part
            - Red is `#ff0000`, green is `#00ff00`, and blue is `#0000ff`
    - `cp.pixels[m:n] = color * (n - m)` sets the color of NeoPixels `m` thru `n-1`
    - `cp.pixels[:n] = color * n` sets the color of the first `n` NeoPixels
    - `cp.pixels[-m:] = color * m` sets the color of the last `m` NeoPixels
    - `cp.pixels[n]` returns the current color of NeoPixel `n`
    - `cp.pixels.fill(color)` sets all NeoPixels to the same color
        - `cp.pixels.fill((255, 0, 0))` makes all pixels red
        - `cp.pixels.fill(COLOR)` sets all pixels to value assigned to `COLOR`
- **Accelerometer**
    - `cp.acceleration` returns the x, y, and z acceleration values as a tuple
    - Assign accelerations to 3 names using `x, y, z = cp.acceleration`
    - `cp.acceleration.x` returns the x acceleration only
    - `cp.acceleration.y` returns the y acceleration only
    - `cp.acceleration.z` returns the z acceleration only
    - `cp.tapped` returns `True` if a tap or taps are detected since the last request
        - Detects a single tap after `cp.detect_taps = 1` has been set
        - Detects a double tap after `cp.detect_taps = 2` has been set
    - `cp.shake(shake_threshold)` returns `True` if a shake that exceeds the threshold argument has been detected since the last request
        - Default `shake_threshold` is 30
        - Minimum `shake_threshold` is 10 (detects smaller shakes)

- **Light, Temperature, Sounds, and Touch**
  - `cp.light` returns the light sensor value (values range from about 0 to 320)
  - `cp.temperature` returns the value of the temperature sensor in degrees C
  - `cp.play_tone(frequency, duration)` plays a tone for a specific duration in seconds
  - `cp.start_tone(frequency)` starts playing a tone for an indefinite amount of time
  - `cp.stop_tone()` stops any currently playing tone
  - `cp.play_file("sample.wav")` plays the file `"samle.wav"` to the speaker (must be a small, 22 kHz or lower 16-bit PCM, mono `.wav` file)
  - `cp.loud_sound(threshold)` returns `True` if a sound above the threshold was detected since the last request (default `threshold` is 200)
  - `cp.sound_level` returns a float of the existing sound level
  - `cp.touch_A1` returns `True` if capacitive touch pad `A1` is touched (can be used with `A1-A6` and `TX`)

7.4 Example Scripts Using `cp`

Since importing the cp module does all of the initialization work for built-in objects on the CPB, scripts are generally easier to write. The following code blinks the red LED on and off every half-second.

Blink the red LED using the cp module

```
from adafruit_circuitplayground import cp
import time

cp.red_led = True
while True:
    cp.red_led = not cp.red_led
    time.sleep(0.5)
```

This script plays a 440 Hz tone to the built-in (and somewhat buzzy) speaker any time the A4 pad is touched. On a perfectly tuned piano, 440 Hz is the note A4. The note C4 (middle C) on a piano is about 262 Hz.

Play a tone when `A4` is touched

```
from adafruit_circuitplayground import cp

while True:
    if cp.touch_A4:
        cp.start_tone(440)
    else:
        cp.stop_tone()
```

This next code block will turn on all of the NeoPixels and make them red if just button A is pressed. They will all be set to blue if just button B is pressed. If neither button is pressed the NeoPixels will be turned off.

Buttons and NeoPixels

```
from adafruit_circuitplayground import cp

# assign some colors to names
OFF = (0, 0, 0)
RED = (255, 0, 0)
BLUE = (0, 0, 255)

cp.pixels.brightness = 0.25

while True:
    if cp.button_a and not cp.button_b:
        cp.pixels.fill(RED)
    elif not cp.button_a and cp.button_b:
        cp.pixels.fill(BLUE)
    else:
        cp.pixels.fill(OFF)
```

The *Mu* editor has the ability to plot values relative to time. To do so, include a `print()` function that contains a tuple with the values to plot as an argument. Turn on the Plotter in *Mu* when the script is running to see the plot. A tuple can have a single value as long as the comma is still included. The following code plots the light level every 0.5 seconds.

Read the light level for plotting

```
from adafruit_circuitplayground import cp
import time

while True:
    # print a tuple, turn on the Mu Plotter to see graphical results
    print((cp.light,))
    time.sleep(0.5)
```

The following script will plot the x, y, and z acceleration values every second. If the individual values are not needed elsewhere, the `cp.acceleration` command could be placed directly in the `print()` function.

Read and plot acceleration values

```
from adafruit_circuitplayground import cp
import time

while True:
    x, y, z = cp.acceleration
    # turn on the Mu Plotter
    print((x, y, z))
    # the next line will do the same thing
    # print(cp.acceleration)
    time.sleep(1)
```

The following code will blink each of the NeoPixels on then off every 0.25 seconds. It will then turn on all of the NeoPixels one at a time with a 0.25 second delay between each. Finally, it will turn off the NeoPixels one at a time starting with the last one that was turned on. This sequence will repeat as long as the CPB is running.

Sequence the NeoPixels

```
from adafruit_circuitplayground import cp
import time

OFF = (0, 0, 0)
ON = (255, 0, 255)

while True:
    for i in range(10):
        cp.pixels.fill(OFF)
        cp.pixels[i] = ON
        time.sleep(0.25)
    for i in range(10):
        cp.pixels[:i + 1] * i = ON
        time.sleep(0.25)
    for i in range(9, -1, -1):
        cp.pixels[i] = OFF
        time.sleep(0.25)
```

A little bit of buzzer music.

Eine Kleine Summermusik (A little buzzer music)

```
# PCB Buzzer Music for the CPB internal speaker

from adafruit_circuitplayground import cp
import time

tempo = 70/1000

beats = [2, 2, 2, 2, 4, 1, 6, 8, 1, 2, 2, 2, 2,
         4, 1, 6, 6, 1, 2, 4, 2, 2, 2, 2, 6, 4,
         6, 2, 6, 4, 6, 8, 4, 2, 2, 2, 2, 4, 6,
         10, 2, 2, 2, 2, 6, 4, 6, 2, 4, 2, 2, 2,
         2, 6, 4, 6, 2, 6, 4, 6, 8]

notes = ['c', 'd', 'f', 'd', 'a', ' ','a', 'g', ' ',
         'c', 'd', 'f', 'd', 'g', ' ', 'g', 'f', ' ',
         'e', 'd', 'c', 'd', 'f', 'd', 'f', 'g', 'e',
         'd', 'c', 'c', 'g', 'f', ' ','c', 'd', 'f',
         'd', 'a', 'a', 'g', 'c', 'd', 'f', 'd', 'C',
         'e','f', 'e', 'd', 'c', 'd', 'f', 'd', 'f',
         'g', 'e', 'd','c', 'c', 'g', 'f']

names = ["c","d","e","f","g","a","b","C"]
frequencies = [262,294,330,349,392,440,494,523]
frequency = dict(zip(names, frequencies))

print()
print('You have been...\n')

for note, beat in zip(notes, beats):
    print(note, end=" ")
    duration = beat * tempo
    if note == " ":
        time.sleep(duration)
    else:
        cp.play_tone(frequency[note], duration)

print('\n')
print(chr(82)+chr(73)+chr(67)+chr(75),end=" ")
print(chr(82)+chr(79)+chr(76)+chr(76)+chr(69)+chr(68))
```